Monitoring the processing of interactive requests on distributed systems

Paul Ashton and John Penny
Department of Computer Science
University of Canterbury

TR-COSC 07/95, July 1995

Monitoring the processing of interactive requests on distributed systems

Paul Ashton and John Penny
Department of Computer Science
University of Canterbury
Private Bag 4800
Christchurch, New Zealand

email: paul@cosc.canterbury.ac.nz

Abstract

Distributed systems, and systems with microkernel architectures, are becoming widely used. A consequence is that the processing required to perform a user request is often spread over many computers, many programs and many processes. To gain an understanding of the processing involved in carrying out a user request, and therefore of what contributes to user response times, it is important to capture information on the contributions of all processes that perform work as a result of the request. Existing monitors (for sequential, parallel and distributed systems) are not well suited to this task, as they provide information on either overall system activity, or on the entire execution of a single program.

Here, we define an *interaction* as a user input and all processing that results from that user input. We define an *interaction network* to be a directed graph representation of all events and activities that occur during an interaction, regardless of the computer, process, or program in which they occur. We present a number of interaction networks constructed from event records obtained by instrumenting the kernel of SunOS. These examples illustrate the value of the interaction network as a tool for understanding and evaluating what happens during an interaction. The examples also help to highlight the substantial differences between the information available from monitoring of interactions and the information available from monitoring of individual programs. The interaction network is based on very general models of an 'interaction' and a 'loosely-coupled distributed system', and should be widely applicable.

1 Introduction

From the time of the earliest interactive systems, measuring response times for user requests has been a key part of performance evaluation. In early systems, it was usually the case that few software components were involved in servicing a user request—often a single application program (running as a single process) and the operating system kernel. Monitors designed to assist improvement of interactive performance for such systems provided information such as execution profiles of programs and per-process accounting records for the various resources managed by the kernel. Tools designed to measure usage of system resources were also valuable for improving response time performance, as well as for improving system efficiency, provided that the processing of interactions was confined to a single system.

Interactive computing today is performed on systems that are many times more complex. Several factors have contributed to a considerable increase in the number of hardware and software components involved in servicing a user request. One major factor is that interactive computing usually involves use of

a *loosely-coupled distributed system* which, for the purposes of this paper, is seen as a collection of separate computer systems (nodes) that communicate and synchronise by passing messages across a communication network (hereafter we will simply use the term *distributed system* but will in all cases imply a loosely-coupled system). A second important factor is the emergence of microkernel architectures, in which the functionality formerly provided by a single (large) kernel is provided by a (much smaller) microkernel and a number of (local and/or remote) servers.

Because of the much larger number of hardware and software components involved, the way in which a distributed system executes an *interaction* (responds to a user request) is much more complex than the way in which a traditional time-sharing system executed an interaction. Performance monitors that show only what happens on a single node have limited value when the processing for each interaction is distributed across a number of nodes. Instead, monitors are required that will show **all** the processing that occurs during a distributed interaction.

In this paper, our major objectives are to introduce the interaction network concept, and to illustrate its value using examples. The *interaction network* is a directed graph representation of what happens during processing of an interaction across (possibly many) nodes of a distributed system. We begin by describing the interaction network, and the models for an 'interaction' and a 'distributed system' on which it is based. We then show that recording interaction networks, which are designed to show what happens during an *interaction*, is fundamentally different from recording executions of distributed *programs*. We discuss how interaction networks can be recorded, and describe a prototype monitor developed to record interaction networks in a SunOS environment. We then use three examples of interaction networks that we have recorded to show:

- That a distributed system can be instrumented to record interaction networks.
- That one can, using browsing and analysis tools on interaction networks, gain a great deal of information about what happens during an interaction on a distributed system. We have used the interaction network for analysis of interactive performance and for visualisation of the processing involved in an interaction. We expect debugging to be another application area for interaction networks.

2 Interactions with a Distributed System

The interaction network idea is based on very general models for both a user 'interaction' and for an 'interactive system'.

2.1 Interactions

We see an interaction as an instance of a <user action, system reaction> pair. Common user actions are key-strokes and mouse events (movement, and button presses and releases). A 'user action' may cause a trivial 'system reaction', such as a reaction that consists of moving a mouse cursor. On the other hand, a user action may require processing distributed across several nodes, processes, and programs, before the system reaction is complete. Our model is sufficiently general to cover all such interactions.

2.2 System Model

Our model of execution on a distributed system is that computation is performed by threads that synchronise and exchange data through shared memory (possible only for threads on the same node), or by message passing. Each *thread* is a 'schedulable unit of flow of control' [4].

This distributed system model is developed in detail in [1]. We show there that this model, on which the interaction network concept is based, is general enough to apply to centralised, network, and distributed operating systems, and to collections of operating systems within any loosely-coupled hardware system.

The examples of recorded interaction networks given in this paper are taken from the single system, consisting of two nodes, that we have instrumented to record these networks. The concept of an interaction network can however be applied on any computer system that is within the scope of our very general distributed system model.

2.3 Interaction Analysis vs Program Analysis

Our work is directed at recording, displaying, and analysing what happens during *an interaction* on a distributed system. Our idea has been that, to improve interactive performance, we must tackle directly the question: 'What contributes to the time taken to respond to any user action?'

Intuitively, we know that the processing needed for the system reaction to a user action is substantially different from the execution of a single program. A 'program' may be designed to participate in many interactions; the shell and the X windows server are examples. On the other hand, servicing any interaction requires complete or partial execution of many different programs. Even the simplest system reaction is likely to need execution of code in the command interpreter, code in the operating system kernel and, possibly, code in various servers. A major advantage gained through analysis of system reactions, rather than of program executions, in a distributed system is that a system reaction includes all server processing that results from the user action.

A many-to-many relationship therefore exists between threads and system reactions. Any system reaction is performed by one or more threads, and a thread performs work for one or more reactions. Threads executing code within servers or within interactive programs are commonly involved in many interactions. Some of the threads that execute work for a system reaction may already exist (an example is a file server thread) while others are created during the reaction.

Analysing what happens during an interaction is, we believe, a more effective approach toward improving **interactive** performance for a distributed system than analysing what happens during execution of a program. If a user is using an X-windows interface, for example, analysis of the interactions initiated by the user is much closer to the user view than analysis of the complete executions of (say) the X server program, and the set of X client programs.

2.4 Identifying the extent of an interaction

Our work is based on recording and analysis of interactions, rather than on recording and analysis of program executions as is done, for example, in IPS-2 [10], DPM [9], DTM [6], JADE [7], Monit [8], the work of Tsai *et al.* [11] and the work of Zernik *et al.* [13]. Those program monitors regard the extent of a program execution as the complete execution of one or more processes and all message passing between those processes, which makes associating an event with a program execution very simple.

Construction of a tool to analyse interactions, however, requires the solution of one basic problem: events occurring in different processes must be associated with the interaction of which they are part. We call this grouping of events into interactions *determining the extent* of an interaction.

To associate any event with the interaction of which it is part, the monitoring software must:

- identify any user action that starts an interaction, and generate an identifier for the new interaction.
- store an interaction identifier with each thread and each message.
- assign the appropriate interaction identifier to a thread when it is created and to a message when it is sent, and re-assign the appropriate interaction identifier to a thread whenever it switches to participate in a different system reaction.
- ensure that all event recorders inserted into code include interaction identifiers in the event records that they produce.

This technique for determining the extent of an interaction could be used to determine the true extent of the execution of a program. Monitors directed at program analysis could in this way be enhanced to record **all** execution carried out on behalf of a program, including processing done by file servers, directory servers, windows servers, and so on.

3 The Interaction Network

Within our definition of an interaction as an instance of a <user action, system reaction> pair, we assume that each user action causes the system to perform a *task* that comprises the system reaction. The processing required for a complete task is represented as a directed acyclic graph, our *interaction network*. Acyclic digraphs have been used by others to represent execution of programs (but not of interactions) in parallel and distributed systems. Examples are the Program Activity Graph (PAG) [10], the integrated process-level execution log (IPEL) [11] and the causality graph [13].

Each vertex of an interaction network corresponds to some 'significant' *event* in the life of the task, important examples being: the user request that initiated the task, creation of a new thread, a thread being added to or removed from a system queue, termination of a thread, a thread acquiring or releasing some resource, a thread sending or receiving a message, a message being added to a queue, or a message being sent along or received from a communication link. While all of the events just listed occur at the operating system level, events of any type (procedure call and return, and user defined events for example) can be recorded in interaction network. Each edge of an interaction network corresponds to the work done by a thread, or to the progress of a message, between a pair of events. A more detailed description is given in [1].

Any event occurs at a particular time, so vertex v_i represents an event that occurred at $time(v_i)$. For our performance work [2], we see the event represented by v_i as putting the associated thread or message into some *state* from $time(v_i)$ until the time of the next significant event, where each state is defined in terms of the resources used or queued for. The length of an edge from v_i to v_j then represents the time (that is, $time(v_j) - time(v_i)$) spent in that state. The total times spent in different states, especially along the critical path (see following subsection), can be used to direct strategies for improving response times.

3.1 The Critical Path

The interaction network was conceived originally as a tool for performance evaluation [2], where it is important to identify the critical path through any interaction network. The *critical path* is defined as the sequence of steps that take the longest time to execute. If one wants to reduce response time, then one must make some change that will reduce the length of at least one edge along the critical path. The definition of a critical path is analogous to that given by Yang and Miller [12] for the critical path through a program activity graph (PAG), but in our case the critical path shows the sequence of events that determine the response time for an interaction.

4 Software Tools

To record interaction networks, we have developed a prototype monitor (INMON) for SunOS 4.0. A brief introduction to INMON is now presented, with more detailed information given in [1].

4.1 INMON—recording

Fragments of code were added to the SunOS kernel to associate a 'current interaction' identifier with every thread and every message (see Subsection 2.3). Also, to record the events from which interaction networks are constructed, 53 probes were added to the kernel. Four main groups of probes can be identified. Probes

in one group detect user actions such as mouse or keyboard input. Probes in the second group record process creation and termination events, while the third (and largest) group comprises probes that record communication events. The fourth group largely consists of probes that record events related to resource (processor, disk, network) use.

A process was added to each Sun to write these event records to a *node log file* for later analysis. The first step towards analysis of interaction networks requires identification of the interaction to which each event belongs. This step involves the merging of separate node log files. Interaction identifiers are used to associate events with interactions in production of individual *task log files*, each of which contains event records that make up a single interaction network.

Accurate measurement of message delays requires good synchronisation of the clocks on each node. High resolution Danzig-Melvin (D-M) clocks [5] were used, and clock synchronisation methods were developed that gave synchronisation to within a maximum error of 140 microseconds [3].

4.2 INMON—Analysis

Of the tools for analysing interaction networks, the most important act on task log files. One tool, `inbrowser`, is used to display a visual representation of an interaction network. This browser can display networks in either *time layout*, in which the Y-coordinate of each vertex is proportional to the time of occurrence of the event that the vertex represents, or in *fixed layout*, in which vertices for successive events are equally spaced in the Y direction. Time layout has the advantage that the display shows the relative times of all events, whereas fixed layout makes it easier to see all events when a group of events occur in very quick succession. The interaction networks presented later in this paper are all in time layout.

When `inbrowser` reads a task log file, it displays the entire interaction network in a window. The user can zoom and pan around the network to inspect any part of it in greater detail. Often, events have occurred so close together that one cannot distinguish them without zooming in on a segment of the network. Other features of `inbrowser` include the ability to: display full information for selected events, highlight the critical path, and perform filtering.

The browser for causality graphs developed by Zernik *et al.* [13] is similar to `inbrowser`, but there are a number of differences. Because Zernik's browser is intended for use with parallel programs, placement of vertices reflects a valid partial ordering of events. `inbrowser` gives the option of vertex placement in time layout or fixed layout (a form of partial ordering), and to date we have found the time layout the more useful.

Another tool, `analyse`, is used to compute statistics on event counts and state durations from one or more interaction networks. `analyse` can produce reports that summarise information, for either process- or message-related activities, at the process level, the node level, or for the whole system. An important output of `analyse` is a breakdown of the time along the critical path into components such as times spent: running on or queued for the CPU; waiting for the completion of a disk request; and in transit on a segment of the communication network. Many of the capabilities of `analyse` are illustrated in the next section.

5 Analyses of Some Interaction Networks

The concept of an interaction network, and the insights which recording and analysis of interaction networks can give into processing on a distributed system, are best demonstrated through examining some of the networks that we have recorded. Three interaction networks will be discussed. For the first two, a command line interface was being used; the 'user action' that triggered the system reaction was the input of a newline at the end of the command. For the third, a GUI was being used; the 'user action' in this case was the release of a mouse button at the end of a window resize operation.

All three interaction networks come from execution of interactions in a distributed system that consisted of two Sun workstations (tui and weka), with D-M clocks installed on both machines. In all three cases,

the user action occurred on tui, both machines were connected to the same ethernet segment, and weka provided Sun NFS file service (for many files) to tui. All processes executed on tui, except for `nfsd` (file server) processes, which executed on weka.

In the subsections that follow, general comments appear first, and are followed by more detailed observations. All comments are based on displays produced by `inbrowser`, and on time values and statistics produced by `inbrowser` and `analyse`.

5.1 Listing a directory

The interaction network in Figure 1 involved execution of `waittest`, a program containing only a call to the Unix library function `system()`. That call asked for the execution of the system program `ls` (list directory).

The command line was read by the C-shell command interpreter (`csh`), which created a new process to execute `waittest`. `waittest` then called `system()`, which created a new process to execute the Bourne shell command interpreter (`sh`). `sh` in turn created a process to run `ls`.

The interaction network shows all of the processes that participated in the task, the periods of their participation, and the interprocess communication that occurred. The interaction network includes part of the execution of `csh`, all of the execution of `waittest`, `sh` and `ls` (all four of which executed on tui), and parts of the execution of three `nfsd` processes (all of which executed on weka, a file server). Multiple periods of execution occurred for each of the `nfsd` processes, because many separate file service requests were made. Such behaviour is typical of server processes. The interaction network captures the client-server communication and the degree of server execution on behalf of the client, an important capability of any distributed system monitor. The interaction network shows clearly the relative times of the client-server requests and replies.

Overall, the interaction network contains 700 vertices and the response time (the elapsed time between the first event in the task and the last) was 3.46 seconds.

5.1.1 Detailed observations

1. The time taken by `csh` before it creates the process to run `waittest` is surprisingly long (1.41s of the 3.46s response time). `analyse` reported that `csh` spent over 1 second waiting for (local and remote) page faults to be serviced, with most of that time occurring before creation of the `waittest` process.

2. `analyse` reported that 28 network file system (NFS) remote procedure calls (RPCs) were made: two to get file attributes, one to lookup a name in a directory, two to read a symbolic link, and 23 to read a data block. A breakdown of RPCs per process can be computed if desired.

3. The creation of three processes is shown in Figure 1. In two cases, where `csh` and `waittest` are the parent processes, a group of events (labelled 1 and 2 in the figure) occur in the parent process shortly after it creates the child process. In the third case, where `sh` is the parent process, no such group of events is seen. The reason is that `sh` uses the `fork` system call, in which a complete copy of the parent process is made and then both parent and child can execute concurrently. `csh` and `waittest` use the `vfork` system call, in which the child process borrows the parent's address space (thereby blocking the parent) until the child starts executing a new program image. The groups of events evident in `csh` and `waittest` represent in each case the processing done by the parent process upon return of its address space.

5.2 Pipelines of processes

The interaction network in Figure 2 is for a task designed to study the way in which two processes communicate through a Unix pipe. The input command was `more` *file* | `wc`. `more` reads the specified file and

writes it to the pipe, and `wc` reads from the pipe and counts the number of characters, words and lines read. The interaction network contains 2141 vertices, and the response time for the interaction was 10.34 seconds.

To give an idea of the capabilities of `analyse`, we summarise some results computed from the interaction network shown in Figure 2. The length of the critical path (that is the response time) was 10.34s, with major components: 3.68s CPU time, 4.01s queueing for the CPU, 0.83s performing disk operations, 0.09s in other process states, 0.59s message queueing time and 1.13s spent by messages in transit on network links.

If the critical path length is subdivided by node, the components are: 7.96s on tui, 0.65s on weka, 0.45s for messages from tui to weka, 1.19s for messages in the opposite direction, and 0.08s for messages between tui and tui. The major reason for the time from server (weka) to client (tui) being much larger than the time in the other direction (despite the number of messages in each direction being the same) is that there were 50 NFS read requests (large messages sent from server to client) but no write requests.

The major resource bottleneck is the CPU (74% of the critical path), and the major machine bottleneck is tui (77% of the critical path). Of the time along the critical path spent on tui, `csh` was responsible for 1.08s, `more` for 6.1s, and `wc` for 0.78s. `more` would seem the best place to start when seeking to reduce the response time of the interaction. Once again, it has taken `csh` a substantial amount of time to start the first process, with much of this time spent waiting for page faults to be serviced.

5.2.1 Detailed observations

1. Messages sent through the pipe from `more` to `wc` are easily seen in the network. The file sent through the pipe was around 250Kb, and because a pipe in SunOS has a capacity of 4Kb, more than 50 messages were required to transfer the file.

2. Figure 2 shows that the file server is accessed by `more` between every pair of writes to the pipe. The reason for this 2:1 ratio is that NFS transfers data in units of 8Kb, whereas the capacity of a pipe is 4Kb. In the inset, `inbrowser` has zoomed in on part of the network, allowing this effect to be seen more clearly.

3. The interaction network in Figure 2 was recorded with readahead disabled. We had previously observed from a network recorded with readahead enabled that no NFS requests were made by `more` after its first write to the pipe, even though `more` was reading a 250Kb file. When enabled, the SunOS 'readahead' mechanism was able to (asynchronously) transfer each page of the file from the server before it was needed by `more`.

4. Several messages can be observed going from `wc` to `more` (in the inset, the fourth message from the top is such a message), even though pipes are a one-way communication mechanism. The messages from `wc` to `more` are control messages, each showing a situation where `more` had been blocked waiting to write to the pipe, and the removal of data by `wc` allowed `more` to continue.

5. Study of this interaction network revealed a shortcoming of the SunOS scheduler. The scheduler favours processes using relatively little CPU time, on the assumption that such processes are interactive or I/O bound, and so deserve priority. While this strategy usually works well, consider the case of 2 processes communicating through a pipe. If the rate at which the writing process writes data differs significantly from the rate at which the reading process reads data, then one process (say L) will spend most of its time waiting to send or receive data and will use much less CPU time than the other process (say H). Because L uses little CPU time, the scheduler gives it priority over H whenever L is runnable whereas, to minimise response time, L should only execute when H is blocked. This behaviour suggests a (minor) flaw in the design of the scheduler—perhaps because the designers were thinking of processes competing for CPU time rather than cooperating, as is the case with processes connected via a pipe.

In Figure 2, `more` is process H (as it contains a much greater proportion of the critical path) and `wc` is process L. Instances of `wc` being given priority over `more` were found using `inbrowser`. This is an example of what can be found by a detailed examination of an interaction network.

5.3 Resizing a window under X

The interaction network shown in Figure 3 results from release of a mouse button, as the last action performed by the user in resizing a window.

Three processes on tui were involved in the resize operation—`X`, `tvtwm` and `xterm`. `X`, the X server, is responsible for interacting with the input and output hardware and for providing other (X client) processes with access to the display and to user input. `tvtwm` is a special type of client—a window manager—that lets users do things like moving and resizing windows. `xterm` is the client whose window has been resized.

While the window resize operation is apparently trivial, the interaction network shows clearly that a large amount of computation and message passing was required. The network consists of 1326 vertices and the response time for the interaction was 3.92 seconds. The task involved 25 disk reads (23 on the critical path) and 10 NFS read operations (8 on the critical path), all required to service page faults. At least 1.51s (39% of the critical path) was thus taken up by paging. Being able to analyse system response for such apparently trivial interactions can be of particular interest, because there is nothing more aggravating to a user than a hesitant system reaction for something that the user expects to happen instantly.

5.3.1 Detailed observations

1. The interaction network shows the communication between `X` and `tvtwm`, and between `X` and `xterm`, that results from the window resize operation. An intense period of message passing can be seen between `X` and `tvtwm`. By zooming in on this region of the network, we found that the direction of messages strictly alternates. This intense period of message passing caused a great deal of context switching, and therefore operating system overhead.

2. During the time covered by the interaction network, the `X` process was involved for brief periods in four other interactions, each of which resulted from some keyboard or mouse input. The first of these can be observed as a small discontinuity (labelled 1) in the column of the `X` process. This example shows a server process being shared by concurrent interactions.

6 Conclusions

In this paper, we have introduced the *interaction network*, a directed graph representation of the processing done to give the system reaction to any on-line user action on a loosely-coupled distributed system. We consider the interaction network to be a powerful tool for the understanding and analysis of interactive processing. By instrumenting the kernel of a version of SunOS, we have demonstrated that it is possible to record the information needed to construct interaction networks. We have presented three interaction networks that we have recorded, to show the wealth of detailed understanding that one can gain by study and analysis of interaction networks.

The interaction network is substantially different from tools that show what happens during execution of a program on a distributed system. First, since users see their interactive dialogues in terms of interactions rather than executions of programs, analysis of interactions is much closer to the user's viewpoint than is program analysis. Second, each interaction network shows execution of *all* programs that participate, in full or in part, in performing the system reaction. It therefore shows activities that may not be recorded when analysing program execution, such as processing done by command interpreters, file servers and window servers. Techniques that we have developed for tracking such activities could be used to enhance existing program monitors.

The interactions for which we have presented interaction networks are of relatively small scale. The interaction network concept is however based on a very general model for an interaction, and on a very general model of a system. We believe that software tools can be developed for recording and analysis of interactions of any complexity, on any system that is consistent with our very general model of a distributed system.

References

[1] Paul Ashton. *The Interaction Network: a Performance Measurement and Evaluation Tool for Loosely-Coupled Distributed Systems*. PhD thesis, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, March 1992.

[2] Paul Ashton and John Penny. Decomposition of interactive response times for loosely-coupled distributed systems. In Gopal Gupta and John Lions, editors, *Proceedings of the 14th Australian Computer Science Conference*, pages 19–1–19–10, Sydney, February 1991.

[3] Paul Ashton and John Penny. Experiments with an algorithm for high-resolution clock synchronisation. In Gopal Gupta and Chris Keen, editors, *Proceedings of the 15th Australian Computer Science Conference*, pages 41–55, Hobart, January 1992.

[4] W. C. Brantley, L. G. Brochard, A. Bolmarcich, H. Y. Chang, K. P. McAuliffe, and T. A. Ngo. Initial experiences with RP3 performance monitoring. *International Journal of High Speed Computing*, 1(4):543–561, December 1989.

[5] P. B. Danzig and S. Melvin. High resolution timing with low resolution clocks and a microsecond resolution timer for Sun workstations. *Operating Systems Review*, 24(1):23–26, January 1990.

[6] D. Haban and D. Wybranietz. A hybrid monitor for behaviour and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[7] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[8] T. Kerola and H. Schwetman. Monit: A performance monitoring tool for parallel and pseudo parallel programs. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 163–174, May 1987.

[9] B. P. Miller. DPM: A measurement system for distributed programs. *IEEE Transactions on Computers*, C-37(2):243–248, February 1988.

[10] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, Sek-See Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[11] J. J. P. Tsai, Kwang-Ya Fang, Hong-Yuan Chen, and Yao-Dong Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–915, August 1990.

[12] C.-Q. Yang and B. P. Miller. Performance measurment for parallel and distributed programs: A structured and automatic approach. *IEEE Transactions on Software Engineering*, 15(12):1615–1629, December 1989.

[13] Dror Zernik, Marc Snir, and Dalia Malki. Using visualization tools to understand concurrency. *IEEE Software*, 9(3):87–92, May 1992.

Figure 1: Interaction network for running `ls` using `system(3)` (dotted edges show the critical path)

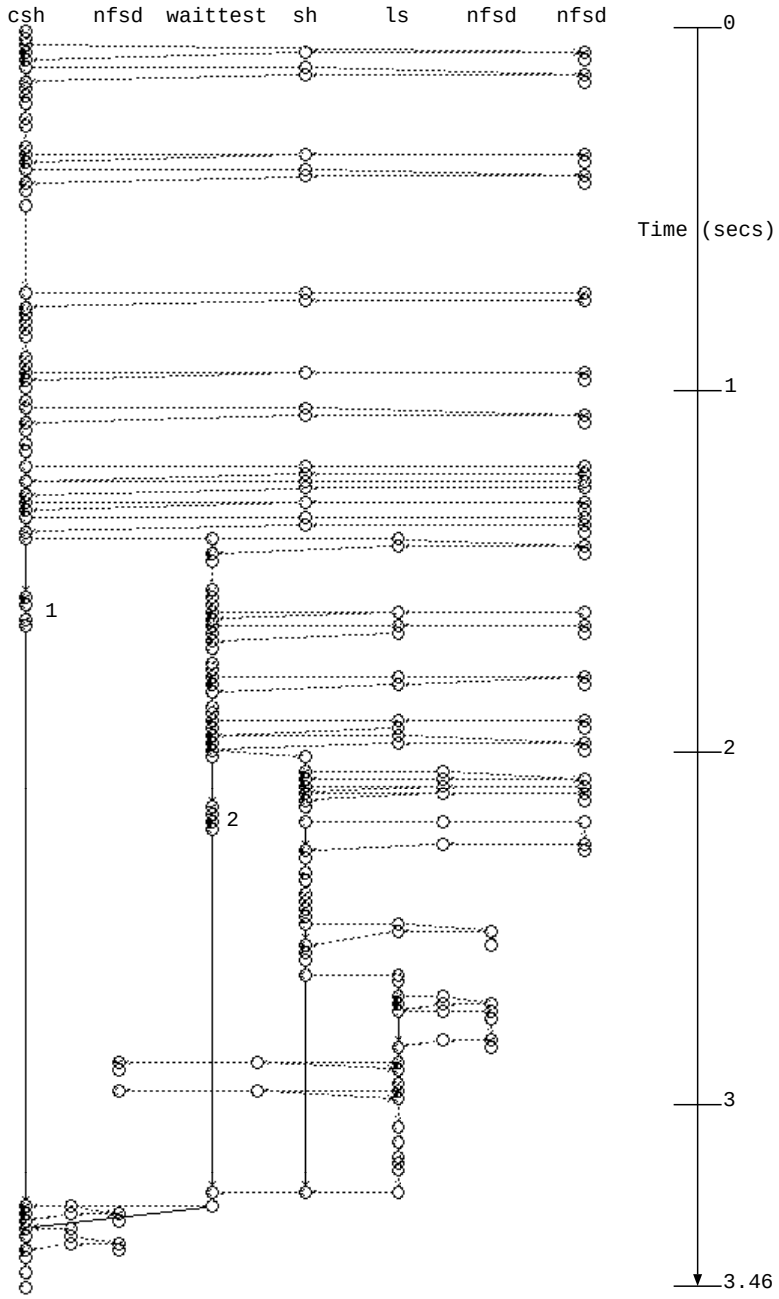

Figure 2: Interaction network showing use of a pipe (dotted edges show the critical path)

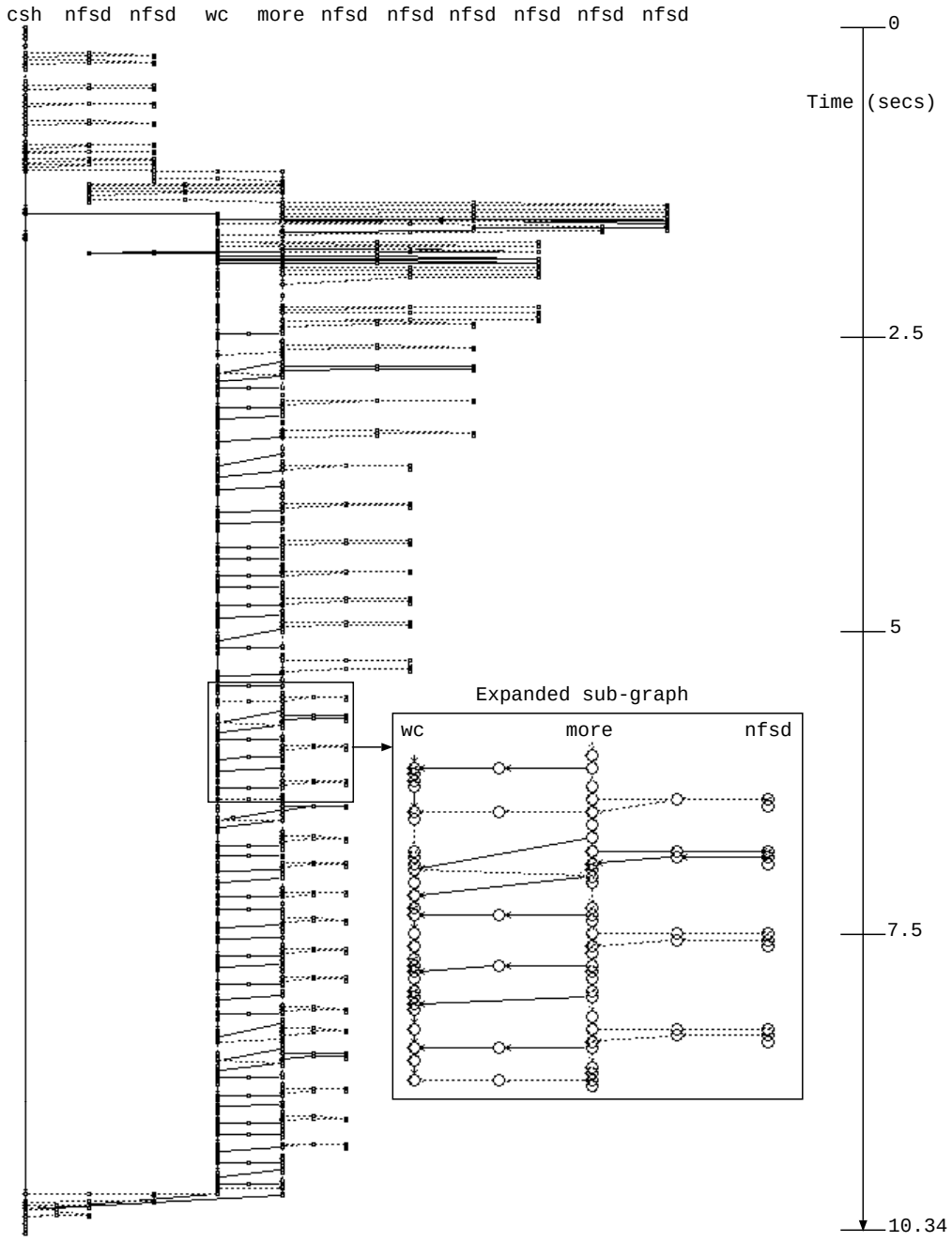

Figure 3: Interaction network for the window resize task (dotted edges show the critical path)